XMOS Architecture XS1 Chips

David May

XMOS

Introduction

Electronic products: design cycles shortening and product diversity increasing

Replace hardware components with software components

Use standard concurrent hardware components to execute diverse concurrent software components

Standard hardware exploits economies of scale in manufacturing

Software supports short design cycles, re-use, diversity, open-source ...

Architecture

An architecture for a range of *concurrent processing* components

Multi-threaded XCore processors connected by *links* and *switches*

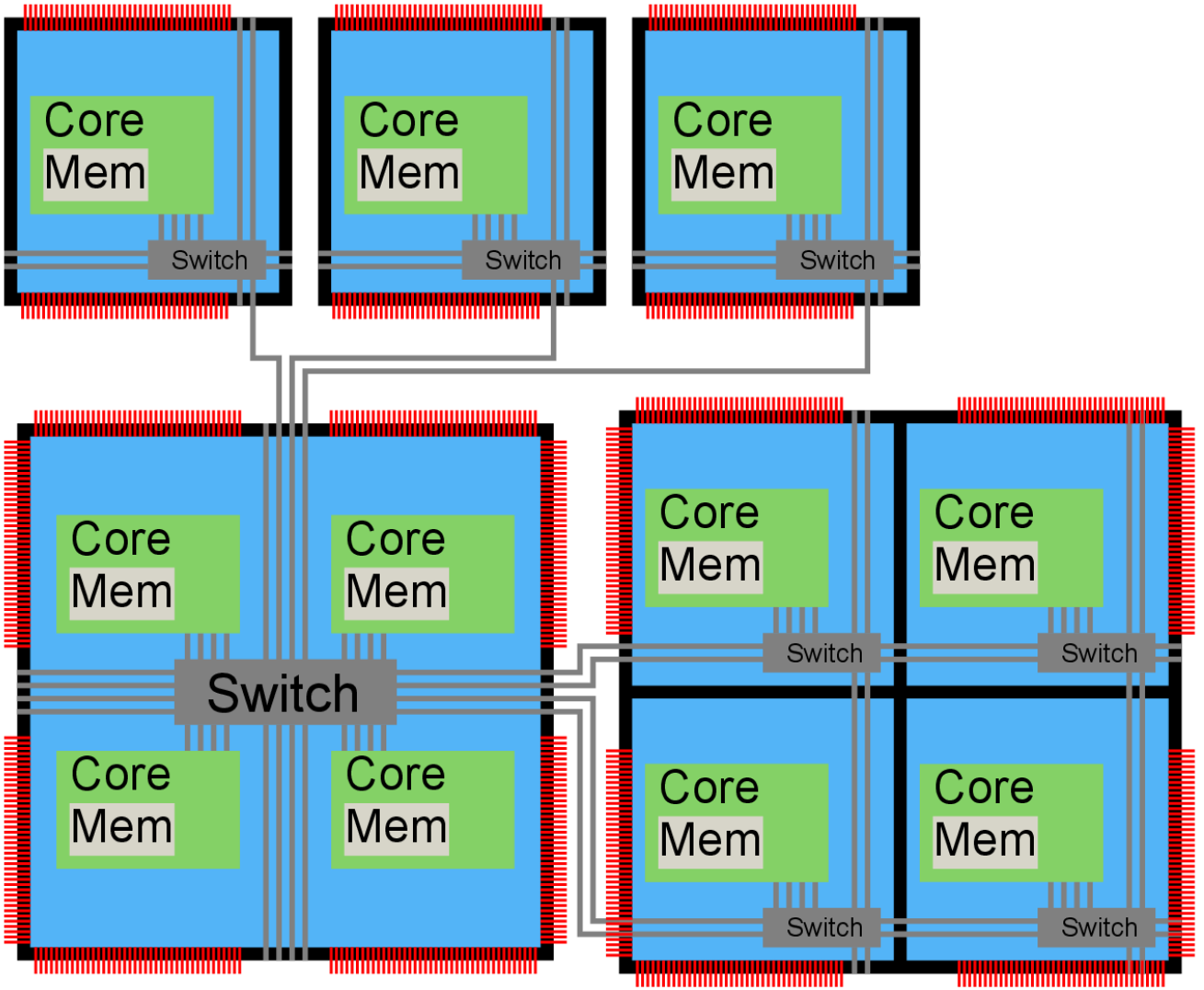
XCores interface directly with external devices via integrated *ports*

Deterministic execution and interface timing

Initial products optimised for embedded applications

Systems built on multi-core chips, in packages or on boards

Example System



Programming

C-based language (along with C and C++) supporting

- deterministic concurrent and multi-core programming
- deterministic real-time and input-output programming

Simple concurrent programming using message-passing

Compiled directly to cores - no kernel or RTOS needed

Real-time performance guaranteed by tools and architecture

An alternative to complex, non-deterministic, cache-coherent shared memory

Scalability

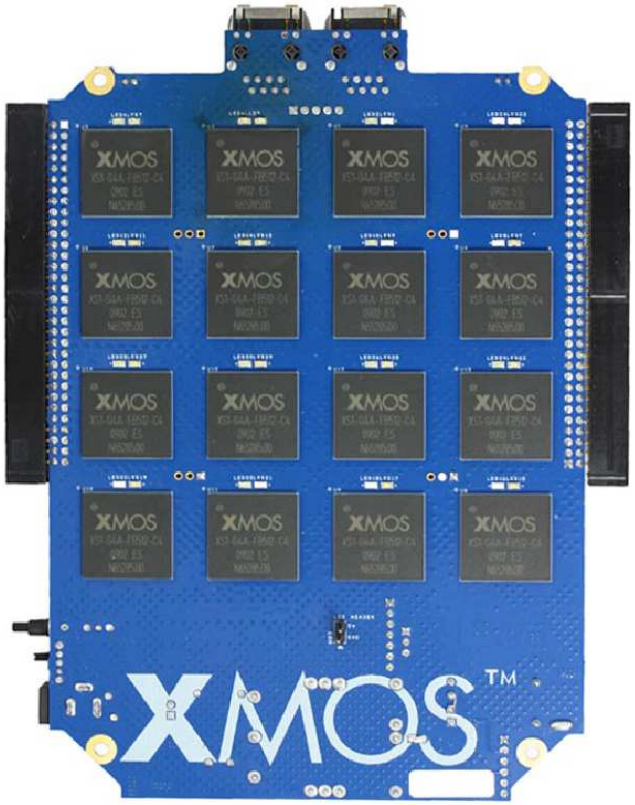
On-core memory, threads, links, ports can all be varied

ISA supports different wordlengths - and has space for new instructions

Switch-based interconnect with scalable throughput

Memory, processing, communication, event-handling scale with cores

From one to hundreds of cores per chip



Threads

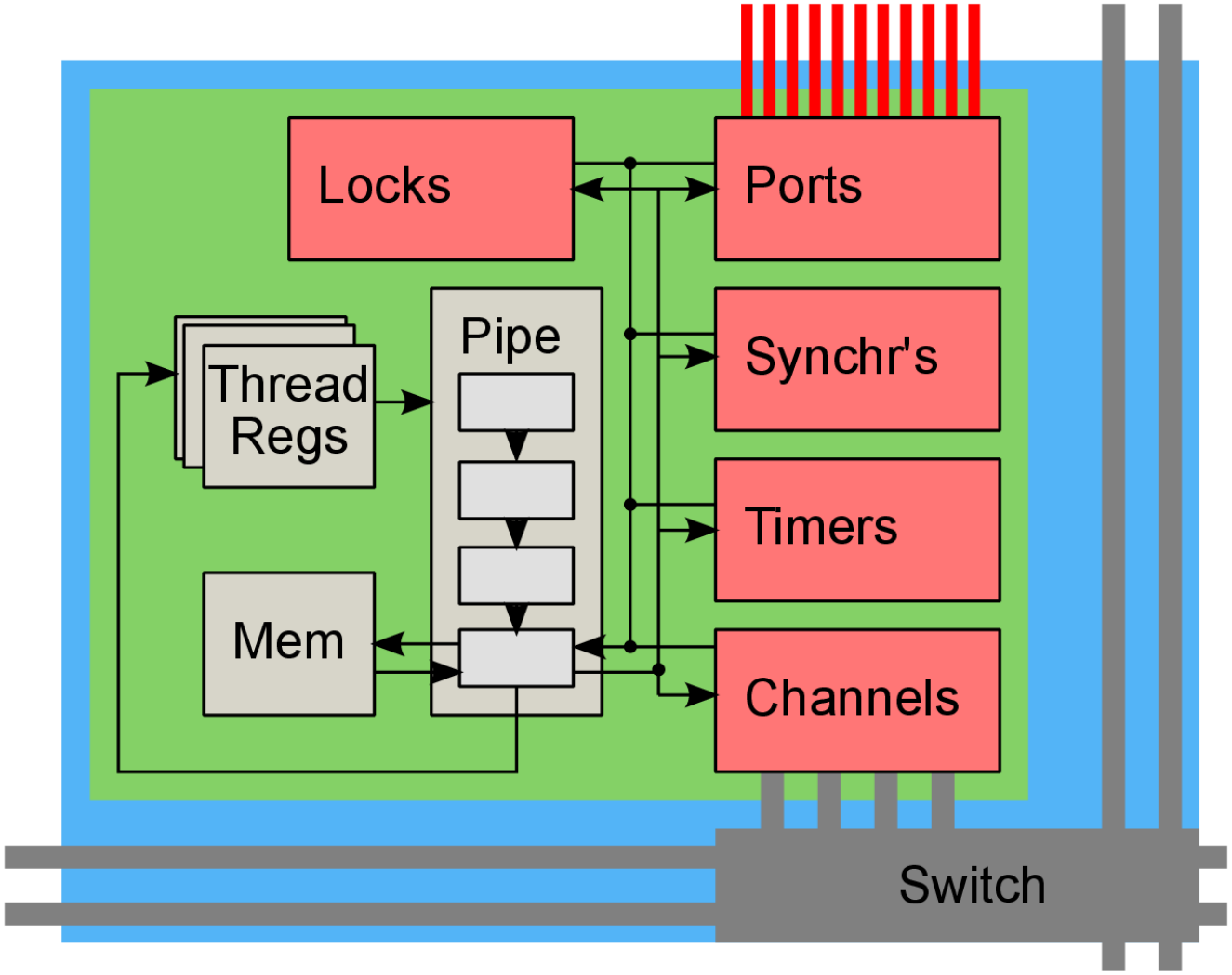
Each XCore provides *hardware* resources for a number of threads, including:

- a set of *registers* for each thread
- a *scheduler* which dynamically selects which thread to execute
- a set of *synchronisers* for thread synchronisation
- a set of *channels* for communication with other threads
- a set of *ports* used for input and output
- a set of *timers* to control real-time execution

Memory for code and data is shared between the threads

Threads are used for latency hiding or to implement ‘hardware’ functions such as DMA controllers and specialised interfaces

XCore Architecture



Instruction set

Each thread has its own register set

Dedicated registers for program counter, stack pointer, data pointer and constant pool

12 general purpose *operand* registers - allowing three operands to be encoded using 11 bits (because $12 \times 12 \times 12 < 2048$) leaving 5 opcode bits.

Most instructions encoded in 16 bits

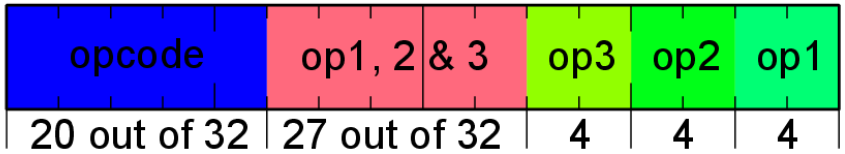
Prefix instructions used to

- extend the immediate range for jumps and offsets
- provide up to 6-operand instructions

XCore Instruction Encodings

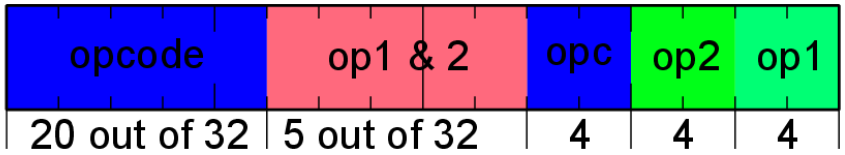
Short 16b Encodings (80%)

3 operand



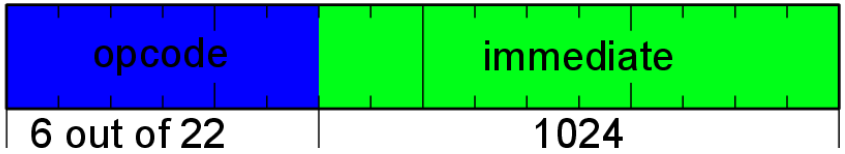
Mostly arithmetic

2/1/0 oper



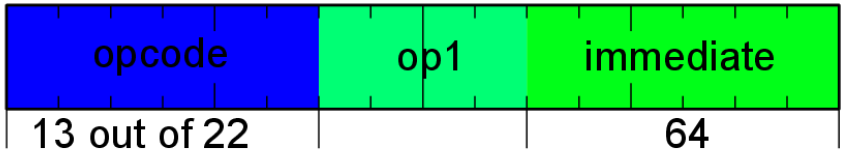
Mostly resources

Big offset



Function calls

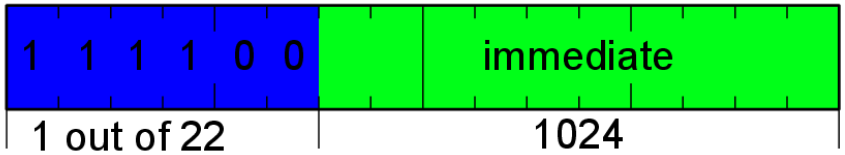
Reg+offset



Branches

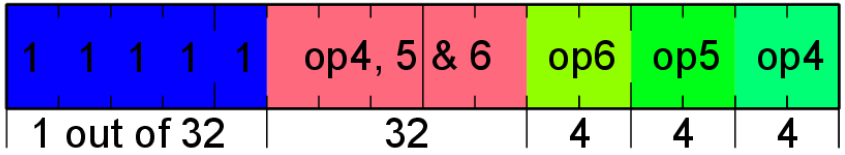
Long 32b Encodings (20%)

ImmPrefix



Immediate extension

InstrPrefix



4, 5, and 6 operand

Thread Scheduler

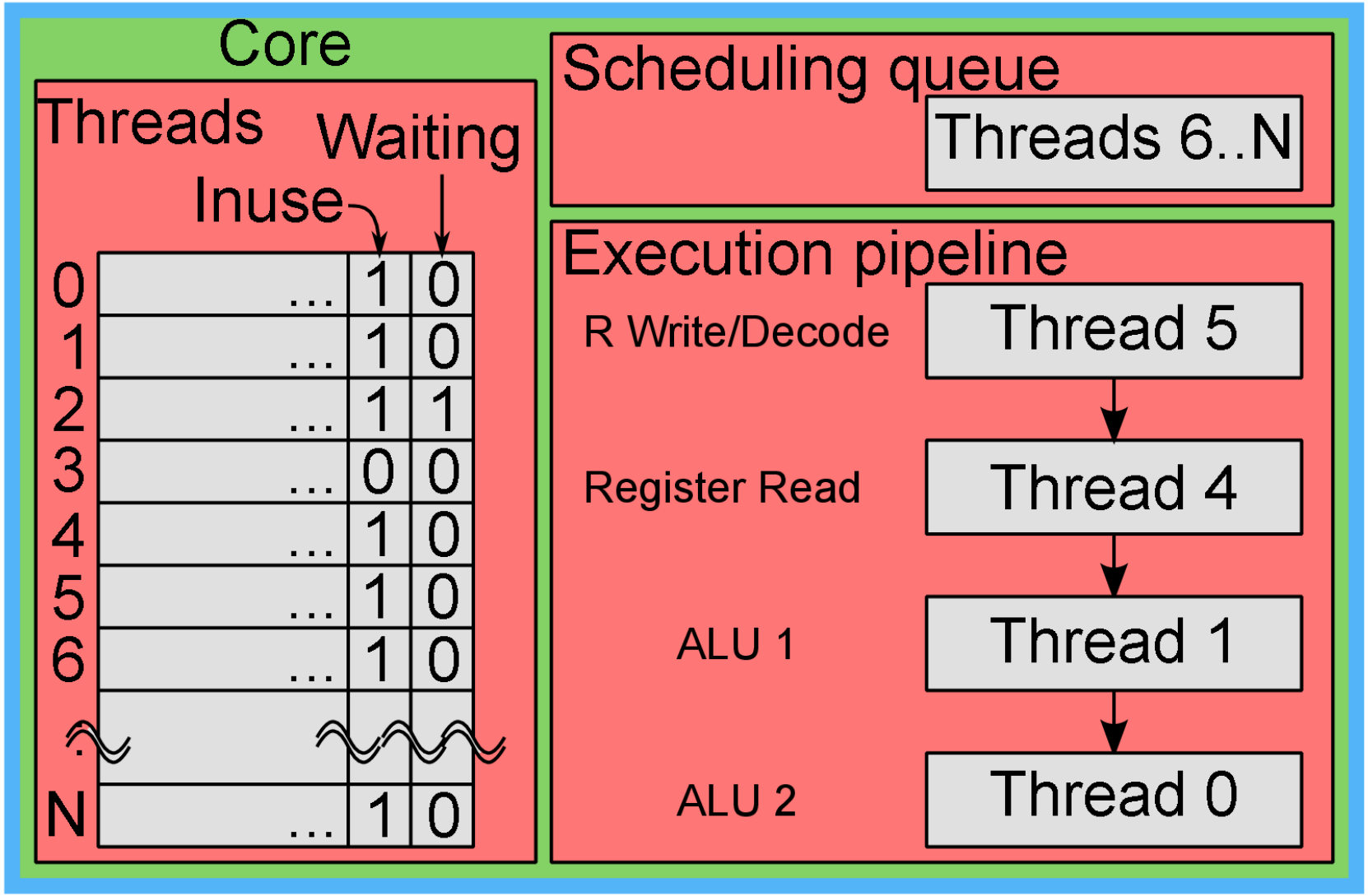
The thread scheduler maintains a set of runnable threads, *run*, from which it takes instructions in turn.

A thread is not in the *run* set when:

- it is waiting to synchronise with another thread before continuing or terminating.
- it has attempted an input but there is no data available.
- it has attempted an output but there is no room for the data.
- it is waiting for a timer.
- it is waiting for one of a number of events.

An XCore can power down when all of its threads are waiting - *event-driven* processing

Thread Scheduler



Concurrency and Thread Scheduler

Fast initiation and termination of threads - forking and joining.

Fast barrier synchronisation - one instruction per thread

Guarantee that each of n threads has $1/n$ core cycles.

A chip with 128 cores each able to execute 8 threads can be used as if it were a chip with 1024 cores each operating at one eighth of the clock rate.

Each core behaves as symmetric multiprocessor with 8 cores sharing a memory with no access collisions and with no caches needed.

Instruction Execution

Each thread has a short instruction buffer sufficient to hold at least four instructions.

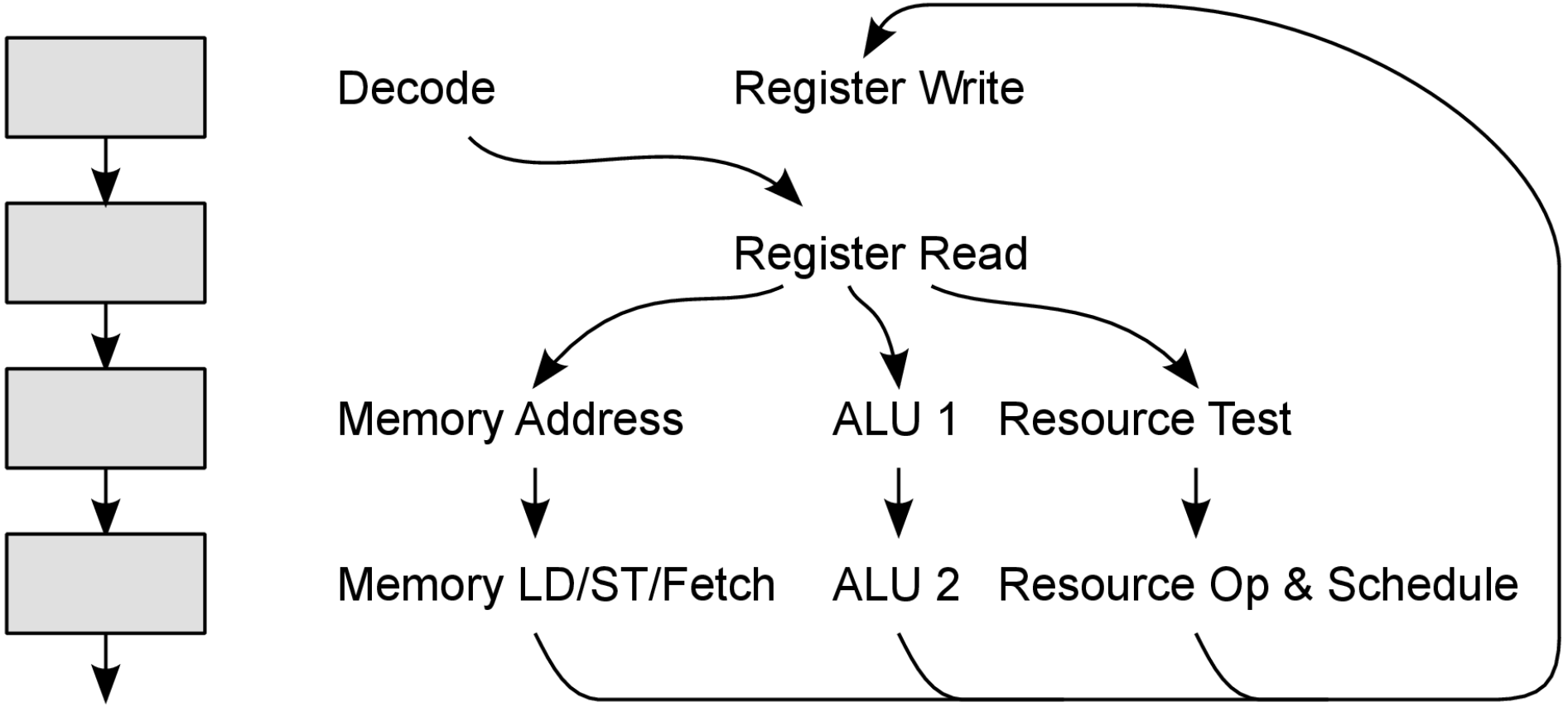
Instructions are issued from the *runnable* threads in a round-robin manner - at most *one* instruction per thread in the pipeline.

Instruction fetch is performed within the execution pipeline, in the same way as data access.

If an instruction buffer is empty when an instruction should be issued, a *no-op* is issued to fetch the next instruction.

Most *no-ops* are eliminated by compiler instruction scheduling.

XCore pipeline



Communication

Communication is performed using hardware *channels*, which provide bidirectional data transfer between threads

- in the same core
- in different cores on the same chip
- in cores on different chips

A channel can be used as a destination by any number of threads - *server threads* can be programmed

The channel addresses are system-wide and can themselves be communicated

Channel protocol provides *control* and *data* tokens; applications optimised protocols can be implemented in *software*.

Channels and Interconnect

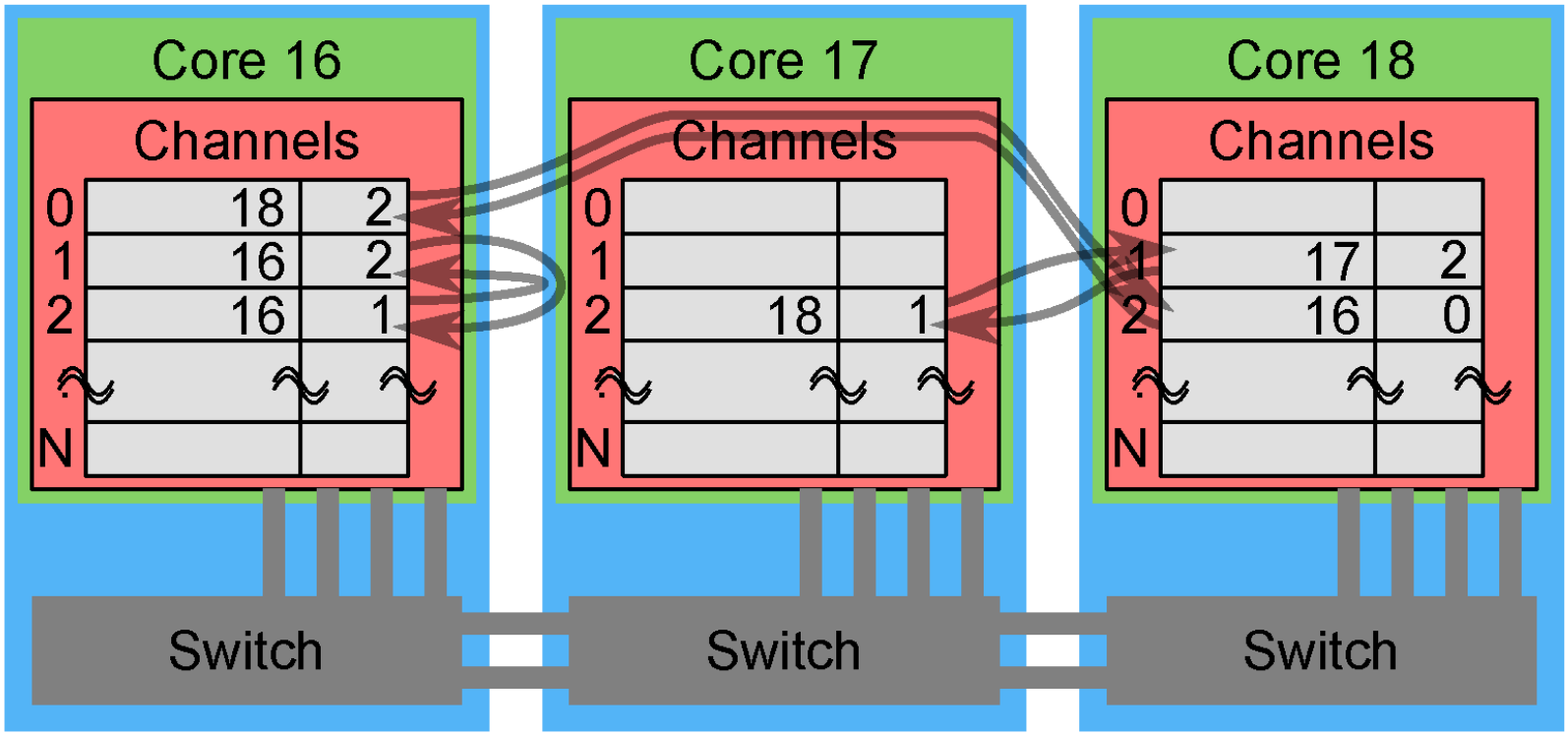
Each core has several bidirectional links to a switch enabling several simultaneous data streams

A route is opened by sending a destination channel address and closed by sending an *end-of-message* token.

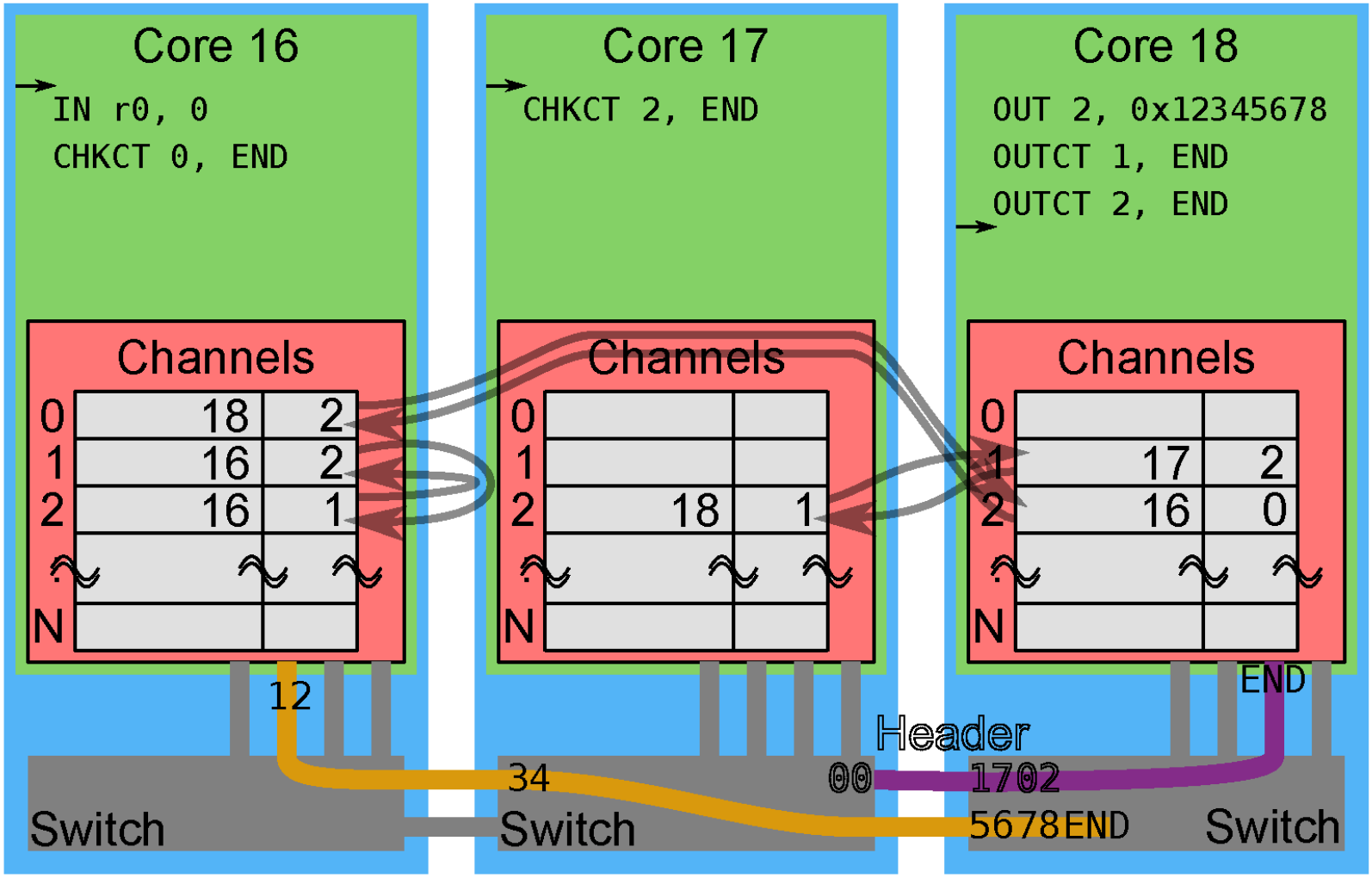
The interconnect can be used under software control to establish *virtual circuits* or perform dynamic *packet routing*.

A set of links can be configured to provide several independent networks - important for diverse traffic loads - or can be grouped to increase throughput

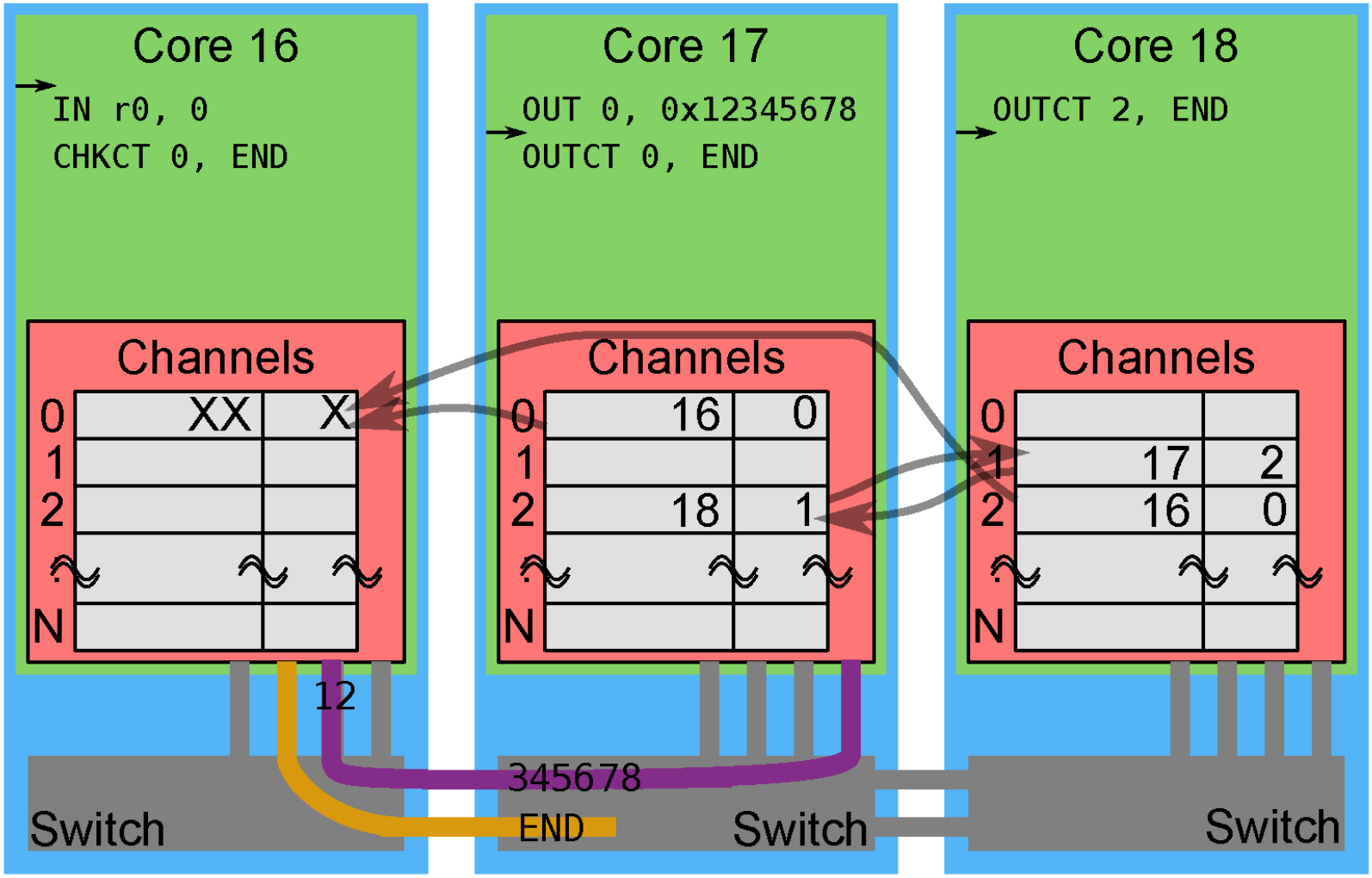
Communication: Addressing



Communication: Messages & Streams



Communication: Sharing



Routing

Simple hardware operating on the first few bits of each message

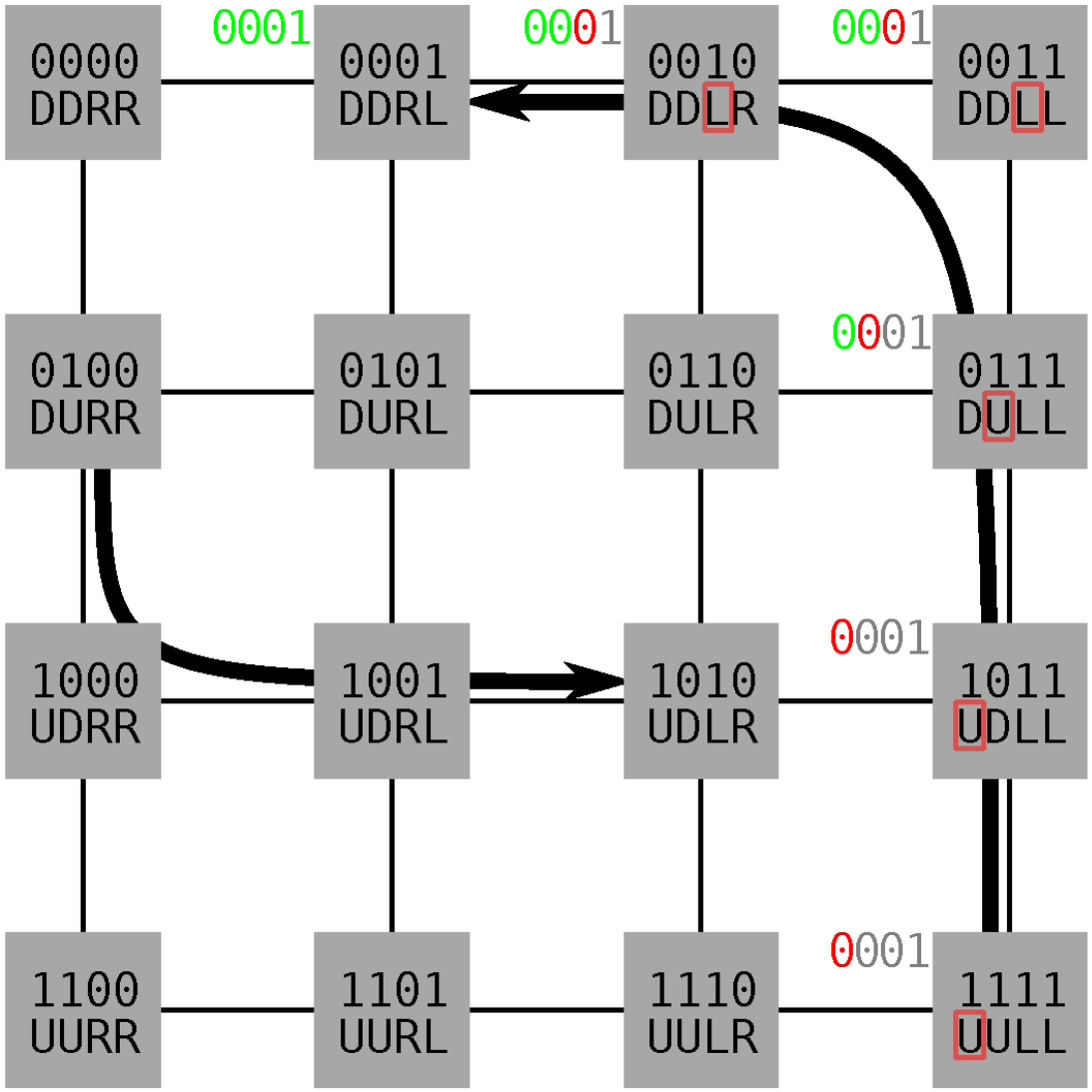
Incoming bits compared with switch address, bit-by-bit

If all pairs match, then a core on this switch is the destination

If not, the number of the first non-matching pair is used to select an outgoing direction from the switch via a lookup table

This is sufficient to perform deadlock-free routing in all n-dimensional grids - and many other structures

Routing Example



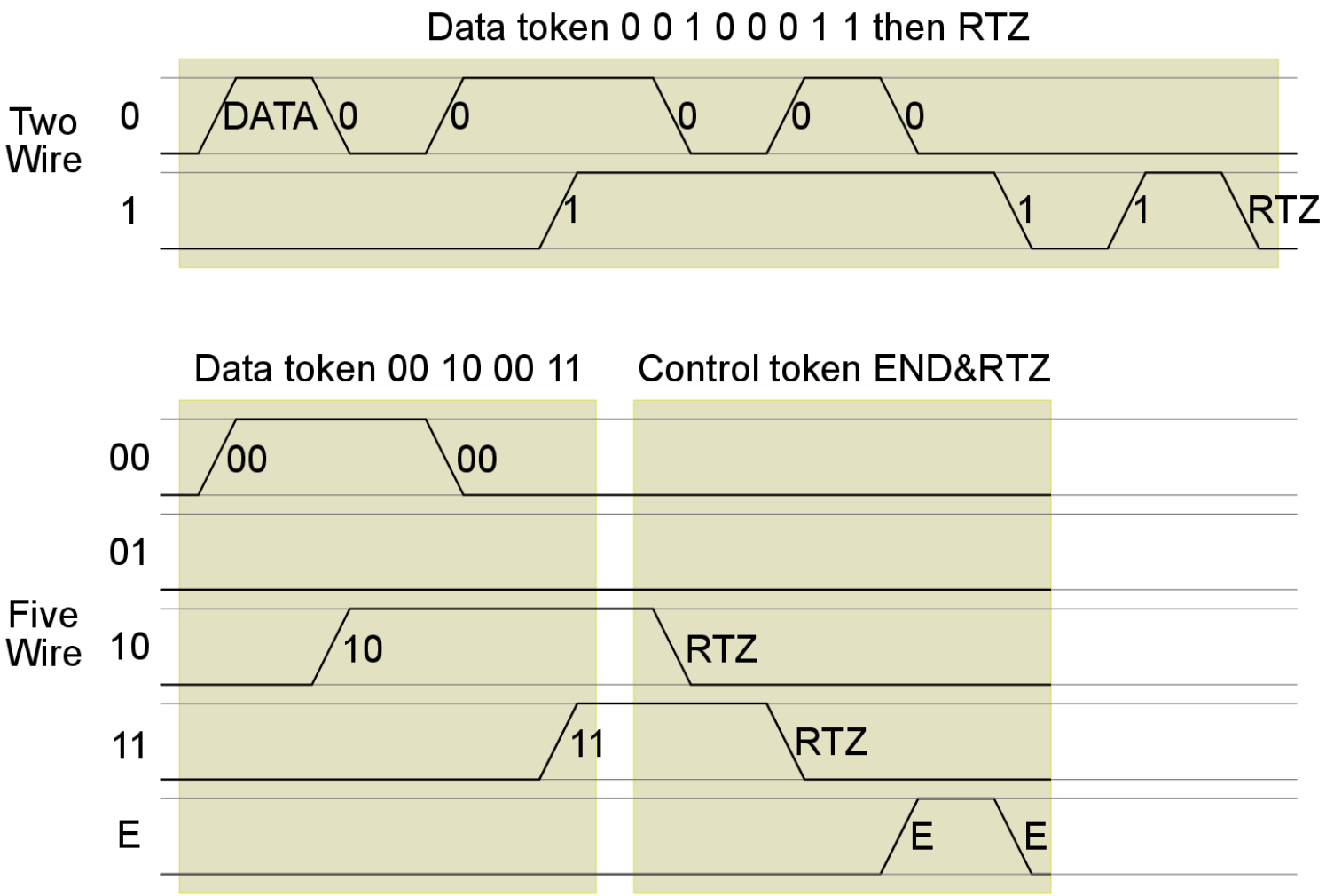
Binary addresses

Leftmost bit mismatch determines next 'direction':

D: Down
U: Up
L: Left
R: Right

Example routes
0100 to 1010
1111 to 0001

Link Protocol



Ports, Input and Output

Inputs and outputs using ports provide

- direct access to I/O pins
- accesses synchronised with a clock
- accesses timed under program control

An input can be delayed until a specified condition is met

- the time at which the condition is met can be *timestamped*

The internal timing of input and output program execution is decoupled from the timing of the input and output interfaces.

Ports and threads can implement ‘hardware’ interfaces

Event-based scheduling

A thread can wait for an event from a set of channels, ports or timers

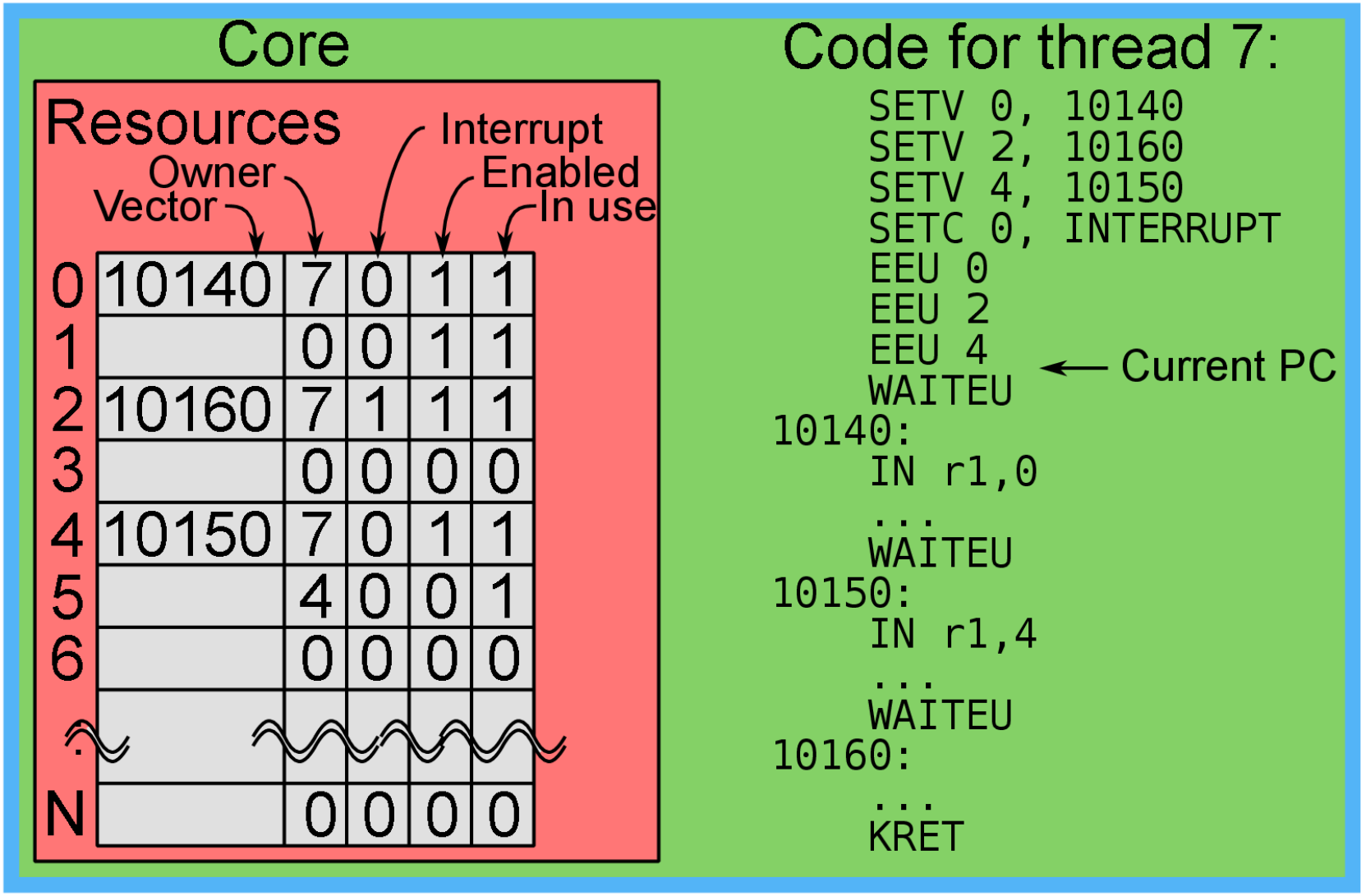
An *entry point* is set for each resource; a *wait* instruction waits until an event transfers control directly to its associated entry point

The data needed to handle each event have been initialised prior to waiting, and will be instantly available when the event occurs

Compilers optimise repeated event-handling in inner loops. The thread is operating as a programmable state machine and events can often be handled by short instruction sequences

This is much faster and more energy-efficient that interrupts.

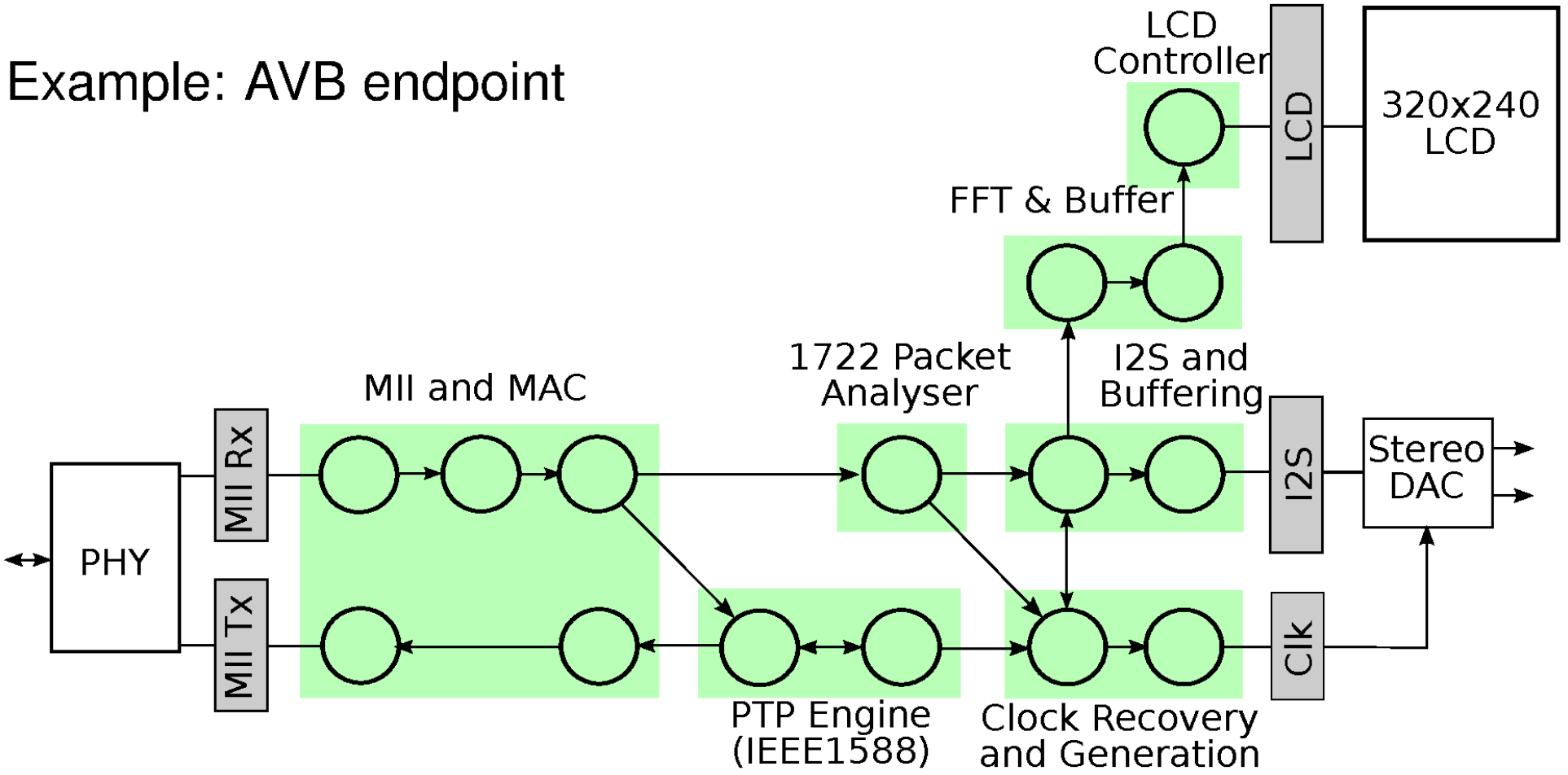
Events


Core
Resources
Owner
Vector
Interrupt
Enabled
In use
0 10140 7 0 1 1
1 0 0 1 1
2 10160 7 1 1 1
3 0 0 0 0
4 10150 7 0 1 1
5 4 0 0 1
6 0 0 0 0
N 0 0 0 0
Code for thread 7:
SETV 0, 10140
SETV 2, 10160
SETV 4, 10150
SETC 0, INTERRUPT
EEU 0
EEU 2
EEU 4
WAITEU
Current PC
10140:
IN r1,0
...
WAITEU
10150:
IN r1,4
...
WAITEU
10160:
...
KRET

Applications

Customers in audio, industrial control, motor control, robotics, vision, and other real-time and embedded domains

Example: AVB endpoint

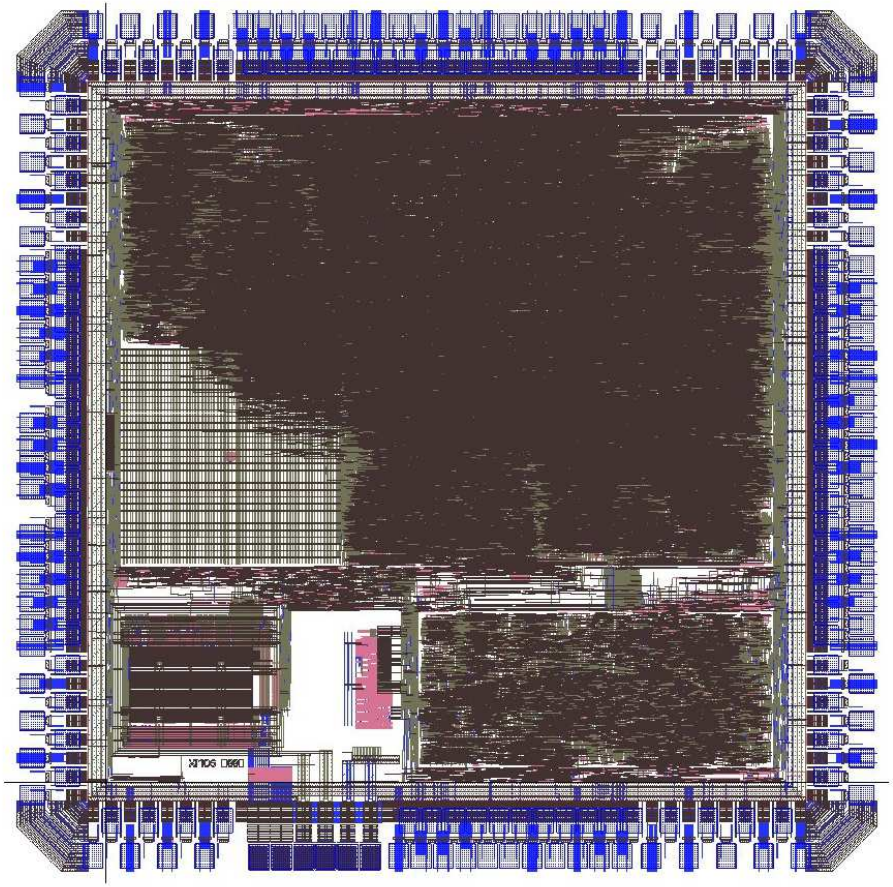


XMOS XS1-G4

Four cores	400 MHz, 1600 MIPS; 32 threads
DSP	$>$ 400 MMACs per second
Events	400 MEvents per second
Switch	4 links per core; 16 external links
Links	16 at 400Mbits/second
SRAM	64k bytes per core
Synchronisers	7 per core
Timers	10 per core
Channels	32 per core
Ports	1, 4, 8, 16, 32-bit
Node	90 nm
Costs	less than $10 in volume

XMOS XS1-L1

One core	500 MHz, 500 MIPS; 8 threads
DSP	$>$ 125 MMACs per second
Events	125 MEvents per second
Switch	4 links for the core; 8 external links
Links	4 at 400Mbits/second
SRAM	64k bytes per core
Synchronisers	7 per core
Timers	10 per core
Channels	32 per core
Ports	1, 4, 8, 16, 32-bit
Node	65 nm
Costs	less than $2 in volume



www.xmos.com